Implementation Of A Generalized Transitive Closure For Relational Queries[*]

M. Mainguenaud, M. Scholl
INRIA, 78153 Le Chesnay CEDEX

ABSTRACT

The use of transitive closure in expressing and evaluating recursive queries becomes central to query processing, the more so as it has been claimed that most recursive queries of practical importance can indeed be expressed by using transitive closure. Because of the iterative nature of evaluation algorithms, it would be nice to have some generalized transitive closures taking into account selection. [SS88] defines such a closure.

The purpose of this paper is to look at efficient methods for computing the generalized transitive closure as defined in [SS88].

We review various strategies based on two algorithms. The first one is a regular graph traversal algorithm. The other one is a new algorithm based on Warshall's one [War62] which provides a very fast transitive closure.

4ème Journée Bases de Données Avancées
Benodet 17.20 Mai 88

[*]This work was supported by the French national program on Databases, PRC BD3.

I. INTRODUCTION

Processing recursive database queries is far from optimal (see for example [BR86, GS87] for comprehensive surveys on this topic or [AJ87], [LMR87], [Sch83]).

However commonly occurring subclasses of recursions can be evaluated efficiently using algorithms specially tailored to these subclasses.

Linearly recursive queries are an example of such subclasses. Among these, one-sided recursions are the simpliest [Nau87].The evaluation of such queries can be expressed as the transitive closure of a binary relation.

We refer to [Nau87] for the definition and characteristics of one-sided recursions. This definition uses the notion of full argument/variable (A/V) graph. Since there is no simple and intuitive definition of a one-sided recursion, we rather give two examples to illustrate this notion. First the transitive closure is the canonical one-sided recursion :

```
t(X,Y)       <-    a(X,Z), t(Z,Y)
t(X,Y)       <-    b(X,Y)
```

the "same generation" problem is a two-sided recursion :

```
sg(X,Y)      <-    p(X,W), p(Y,Z), sg(W,Z)
sg(X,Y)      <-    sg0(X,Y)
```

but the following recursion is one-sided :

```
t(X,Y,Z)     <-    t(X,U,W), e(U,Y), d(Z)
t(X,Y,Z)     <-    t0(X,Y,Z)
```

Then the use of transitive closure in expressing and evaluating recursive queries becomes central to query processing, the more so as it has been claimed that most recursive queries of practical importance can indeed be expressed by using transitive closure. Therefore several authors attack the problem of extending relational languages by including a transitive closure operator (e.g. [Agr87]).

In principle, transitive closure should not be restricted to linearly recursive queries. Because of the iterative nature of evaluation algorithms, strategies should be found to translate any recursion into some transitive closure. In practice, this is not the case since queries always introduce some selection operator and since one cannot in the general case transpose the selection and the closure operator. It would be nice to have some general transitive closure taking into account selections.
As an example, take the relation :

FLIGHTS (SOURCE, D_TIME, DEST, A_TIME)

and the query :

"Find all flights from New-York to Roma"

This query typically requires that if the flight has at least one stop, say in Paris, the arrival time in Paris be less than the departure time from Paris. Clearly no one-sided rule can express such a query. However, there exists an operation, the transitive closure of which can express the above query : [SS88] generalizes the transitive closure of binary relations to an operation for n-ary relations.

A generalized composition operator is defined on relations R1 and R2 in $D_1 x D_2 x ... x D_N$ as follows [SS88] :

$$g(R1, R2) = \Pi_P (\sigma_F (R1 x R2))$$

where σ_F selects tuples in the cartesian product R1xR2 according to the condition F and Π_P projects columns from R1xR2 such that the result has same degree as R1 and R2.

A very general binary operator is thus defined, the transitive closure of which may be computed by using efficient methods such as those currently developped for computing the transitive closure of a binary relation [Lu87].

The purpose of this paper is to look at efficient methods for computing the transitive closure of g. We restrict our attention to the case where R1 and R2 are instances of the same relation R although the algorithms should be extensible to the general case.

In the above query, all possible sequences of flights are defined by :

$$FLIGHTS^+ = \bigcup_n FLIGHTS^n$$

where

$$FLIGHTS^n = g(FLIGHTS^{n-1}, FLIGHTS) \qquad n > 2$$

and

$$FLIGHTS^2 = \Pi (\sigma_{DEST1 = SOURCE2 \wedge D\_TIME2 > A\_TIME1} (FLIGHTS \times FLIGHTS))$$

The objective of the paper is two fold :

1) We first give a new algorithm which allows 1) to provide the regular transitive closure of binary relations as well as 2) keeping track of the intermediate steps (e.g. keep information on intermediate connections in flights from L.A. to Roma). This algorithm (section 3) is based on Warshall's algorithm [War62] and efficient graph traversal. It provides

a) an extremely fast Transitive Closure as Warshall algorithm does

b) as well as complete paths in the graph associated to the transitive relation*.

As a matter of fact to get complete paths, only a subgraph is traversed and this subgraph traversal is much faster than that of regular graph traversal algorithms since it does not necessitate any backtracking. We then study the behaviour of this algorithm in the case of cyclic relations (section 4).

2) the second objective of this paper is to address the problem of finding strategies for efficiently implementing a generalized transitive closure operator (section 5). We compare two basic operators for such strategies :

a) the algorithm defined in section 3

b) a regular graph algorithm, such as that reported in [Puc87, Agr87].

*This graph will be defined in section 2. For the time being, assume that, to each tuple in relation FLIGHTS, corresponds a directed edge labelled by the subtuple < D_TIME, A_TIME> value and whose origin (target) node is labelled by the attribute SOURCE(DEST) value.

The major result of this comparison (section 6) is that the algorithm of section 3 performs well as long as the selection σ_F implied by the operator g does not alterate significantly the graph. This is typically the case of the above FLIGHTS application. Indeed in the graph associated to the recursive relation, the number of edges corresponding to a given pair of nodes can be very large (e.g. there are a large number of direct connections between Paris and New-York) and there is a small chance that a given selection deletes this edge.

However, the new algorithm, at least in its current implementation, requires a precomputation whose predicted worstcase time complexity is prohibitive for transitive relations of large cardinality.

We first give some definitions :

II. DEFINITIONS

The generalized composition operator introduced above [SS88] is defined on relations R1 and R2 as follows :

$g_{P,F}(R1, R2) = \Pi_P (\sigma_F (R1xR2))$

Let us restrict our attention to the case where R1 = R2 and let us denote $g_{P,F}$ (R, R,) by g (R) (or g when there is no ambiguity). We further assume that the selection formula F includes one condition $A_2 = B_1$ where A and B are attributes of R defined on the same domain : $D_A \cup D_B = D$, whose cardinality is denoted by n, and A_2 (B_1) is the second (first) instance of attribute name A(B) in RxR. We are interested in the transitive closure of the following (mathematical) relation :

$$x \rho y \text{ iff } (x,y) \in \Pi_{AB}\, g(R)$$

R is called a (generalized) transitive relation. Tuples of relation R are drawn from :

$D \times D \times D_1 \times \ldots \times D_n$.

II.1 Graph associated to a transitive relation

Then to relation R and to attributes A ,B, one can associate a labelled directed graph G (R, A, B) referred to simply as a graph and defined as the following ordered 4-tuple :

$(N, \varphi_N, E, \varphi_E)$

where N = {1,2,...n} is the set of nodes, φ_N is a one-to-one node labelling function that associates to each node a distinct value drawn from D (the set of values taken by A or B in R), E is the set of edges and φ_E is an edge labelling function which associates with each tuple $abd_1...d_n$ of R a directed edge from node $\varphi_N(a)$ to node $\varphi_N(b)$.

There are no isolated nodes and of course to a given edge there may correspond several tuples in R.

For the sake of simplicity, φ_N domain has been taken as a single domain D. However there is no restriction on the node labelling function:

One may extend the above definitions to the case where A and B are replaced by $A_1, \ldots, A_p$ and $B_1, \ldots, B_p$ defined on $\Delta_1, \ldots, \Delta_p$. Then to each node $i \in N$, there corresponds a unique value from $\Delta_1 \times \ldots \times \Delta_p$:

$$\varphi_N : \Delta_1 \times \ldots \times \Delta_p \rightarrow N$$

where R is defined on :

$$\Delta_1, \ldots, \Delta_p, \Delta_1, \ldots, \Delta_p, D_1, \ldots, D_n$$

II.2 Transitive closure

We are interested in the transitive closure of relations R as defined above. If R (A, B,...) is a transitive relation, its transitive closure R^+ is defined by :

$$R^+ = \bigcup_{i>0} R^i$$

where R^i denotes the ith power of R :

$R^1 = R$ and

$R^i = g(R^{i-1}, R)$ for $i > 1$

Then a pair (a, b) where $a \in \Pi_A(R)$, $b \in \Pi_B(R)$ is in the transitive closure whenever there is a path of non zero length from a to b in the graph G(R, A, B).

II.3 Matrices

A simple and common way of representing a graph is the use of its transition matrix (adjacency matrix). Let us denote by N(i,j) the transition matrix of graph G (R, A, B) defined as an n^2 matrix such that : N(i,j) = 1 if there is an edge from node $i \in N$ to node $j \in N$, otherwise N(i,j) = 0.

We furthermore define the following "reachibility" matrix as follows :

M(i,j) is an n x n matrix such that :

M(i,j) = {k | $k \in N$ and there is a path from i to j in which k is the immediate successor of i}

M(i,j) = $\emptyset$ if there is no path from i to j.

II.4 Example

Let FLIGHTS (S, D, DATE, D_TIME, A_TIME) represent flight connections between cities where S is the source town, D the destination town, and D_TIME (A_TIME) is the departure (arrival) time from city S (at city D).

There are, of course, a large number of direct connections between towns according to the date and the hours.

The query :

"Find the flights from New-York to Roma on January 1st"

should satisfy the constraint that the departure time must be larger than the arrival time in intermediate connection towns (e.g. Paris). This query can be expressed as $FLIGHTS^+$ where the g operator is defined as :

$$g(R) = \Pi_{S1,D2,DATE,D\_TIME1,A\_TIME2}\,(\sigma_{D1=S2 \wedge DATE=JAN1 \wedge D\text{-}TIME2 > A\_TIME1}\,(RxR))$$

G(FLIGHTS,S,D) has towns for nodes and the edges are labelled by DATE, D_TIME, A_TIME values.

III. A TRANSITIVE CLOSURE ALGORITHM

Most algorithms for implementing recursive rules and particularly the transitive closure, use as basic steps, operators of the relational algebra, more specifically the join operator [Lu87]. The problem is then to define the best strategies, a good strategy being not necessarily that which decreases the number of joins, but also that which decreases the size of intermediate results.
However, having in mind the extension of relational algebra with fast transitive closure operators, a few authors look at the design of performant algorithms based on dedicated representations of relations [Puc87, Agr87]. Among those [Puc87] represents a relation by means of its associated graph as defined in the previous section. This work uses an adjacency structure which is equivalent to the transition matrix and shows that such a structure is well adapted not only for computing the transitive closure but also for computing more general linear recursive queries.

The simple idea behind such an approach is to compute from the transition matrix all nodes reachable from a given node by a graph traversal algorithm. This traversal can be done "breadth-first" or "depth-first" according to the query. For example, to a query with aggregate functions+ (applied on the labels of the edges traversed) [RHD86], should correspond a depth-first traversal.

Even for a query without selection condition nor aggregate function calculus, such algorithms imply backtracking steps which for some transitive relations may be extremely large.
There are two causes for backtracking when computing all paths from node $i \in I \subset N$ to node $j \in J \subset N$.
1) backtracking occurs when one eventually reaches an end-node which is not desired : $j \not\in J$
2) backtracking also occurs when one comes to a node already visited (cyclic data).

An alternative is the use of algorithms such as Warshall's one [War62] which permits by a single matrix access to know whether node j is reachable from node i : after applying Warshall's algorithm to N(i,j), the interpretation of the matrix is :

$N(i, j) = 1$ if there is a path from i to j, otherwise

$N(i, j) = 0$.

However, with both approaches, one does not memorize any information on the paths when going from node i to node j, or one does not keep track of the edges and nodes visited (except for some aggregate calculus).

+ Aggregate functions are understood as functions allowing to choose one (or several) path(s) among all paths between two nodes in the graph.

Such an information may be necessary for several applications. As an example, one might require information about, New-York, Paris, ..., which are intermediate stops when travelling from Los Angeles to Roma.

The algorithm presented below is an extension of Warshall's algorithm which allows :

1) As Warshall's algorithm does, to get the transitive closure by a single access matrix and thus without any graph traversal.

2) To traverse more efficiently a subgraph in order.
   a) To compute aggregate functions,
   b) to memorize information on node and edges visited.

This traversal is more efficient, first because olny a subgraph is traversed, secondly because all nodes eventually reached are desired nodes: In that sense there is no backtracking. Of course some backtracking is still necessary when we are in the presence of cyclic data, but we show in section IV, how some permutations on $\{1,...,n\}$ can reduce the number of cycles to be detected.

III.1 Algorithm A

```
For $j = 1$ to $n$, do
    For all $i \in \{1,n\}$ such that $i \neq j \wedge M(i,j) \neq \emptyset$, do
        For all $k \in \{1, n\}$ such that $k \neq j \wedge M(j, k) \neq \emptyset$, do
            $M(i, k) \leftarrow M(i, k) \cup M(i, j)$
        end k
    end i
end j
```

<u>Description</u> :

Algorithm A takes as an entry the n x n matrix $M(i, j)$, where:

$M(i, j) = \{j\}$ if there is an edge from i to j,
$M(i, j) = \emptyset$, otherwise

Then it scans all nodes, and for each of them (column j) looks at all nodes (line i) for which a path has been detected (from i to j : $M(i,j) \neq \emptyset$).
All nodes (k) reachable from j are then detected as reachable from node i through j. Therefore if $l \in M(i,j)$ is the first successor of i on a path from i to j, l is also the first successor of i on a path from i to k through j (elementary step: $M(i,k) \leftarrow M(i,k) \cup M(i,j)$.).

III.2 Example

Figure 2 displays the matrix state after each column has been scanned by Algorithm A for the graph depicted in Figure 1. Step 0 represents the initial state of the matrix.

$$\begin{tikzcd}
2 \arrow[rr] & & 5 \arrow[rr] \arrow[dl, dash] \arrow[dr, dash] & & 1 \\
& 4 & & 3 \arrow[ll] &
\end{tikzcd}$$

Figure 1 an example of graph

	1	2	3	4	5
1					
2					5
3				4	
4					5
5	1		3		

STEP 0

	1	2	3	4	5
1					
2					5
3				4	
4					5
5	1		3		

STEP 1

	1	2	3	4	5
1					
2					5
3				4	
4					5
5	1		3		

STEP 2

	1	2	3	4	5
1					
2					5
3				4	
4					5
5	1		3	3	

STEP 3

	1	2	3	4	5
1					
2					5
3				4	4
4					5
5	1		3	3	3

STEP 4

	1	2	3	4	5
1					
2	5		5	5	5
3	4		4	4	4
4	5		5	5	5
5	1		3	3	3

STEP 5

Figure 2 M(i,j) after each step of algorithm A

III.3 Complexity

The complexity of Algorithm A depends on its implementation. If M is to be implemented by a n^2 matrix where each matrix node M(i,j) is a n-bit vector where the lth bit of M(i,j) is set to 1 if l is the immediate successor of i on a path from i to j (else it is set to 0), then the space complexity is $O(n^3)$ bits.

A simple implementation of algorithm A consists

1. in scanning the n columns, for each column :

2. in scanning the n lines, for each line :

3. in testing whether $M(i,j) \neq \emptyset$ (this can be done by adding to the n-bit vector of M(i,j), a n+1st bit which is set to one if $M(i,j) \neq \emptyset$),

4. If $M(i,j) \neq \emptyset$, then in testing for each column k, whether $M(j,k) \neq \emptyset$. In the latter case,

5. in performing the union of the two n-bit vectors M(i,k) and M(i,j).

Therefore steps 3, 4, 5 are performed n^2 times. The length in time of steps 3, 4, 5 depends on the graph. In the worst case (fully connected graph) it is $O(n^2)$ bit-unions. Then, in the worst case, the time complexity is $O(n^4)$ bit-unions.

<u>A more realistic upperbound on time complexity</u> :

However, for (almost) all applications, the outdegree of any node (e.g. the number of direct connections from a given town in the example above) eventhough it is not bound, is far from its upperbound n.
Then, when n is large, instead of representing M(i,j) by an n-bit vector, it is far better to represent it as a chain of k-bit vectors where k could be the size of, say, a machine word.
Then, if we denote by m the number of edges, the following is an approximation on the upperbound on time complexity :

$O(m/n \times n^3) = O(m \times n^2)$ bit-unions

Indeed, it assumes as above that for each line scanned, $M(i,j) \neq \emptyset$ (step 3), and for each $k \in [1,n]$ there is a test $M(j,k) \neq \emptyset$ which is positive and is followed by m/n bit-unions. Recall we assumed the outdegree to be large, therefore m/n + 1 is nearly m/n.

Observe, that as the previsions $O(n^4)$ worst case upperbound, it is probably far from the average case performance, since it relies on drastic assumptions on the result of tests of M(i,j) (step 3), and M(j,k) (step 4).

However in many applications, the outdegree of any node can be considered as bounded. A less restrictive assumption which corresponds to most applications, would be to consider that m/n is bounded.

Then, a more realistic upperbound on time complexity would be $O(n^3)$ bit-unions.
As an example, take n = 1000 and assume an elementary step (bit-union of two words + overhead) takes 1 micro-second. Then the $O(n^4)$ worst case upperbound gives 10 hours! The $O(n^3)$ worst case upperbound gives olny 16 mn. However, the latter bound gives, with n = 10000, 11 days!
Clearly, a more thorough analysis or a measurement experiment for typical applications is necessary to give a clear idea of the average case time complexity, and to infirm or confirm the worst case prediction which renders this algorithm not realistic for transitive closure relations with large cardinality.

Besides, the memory size must be sufficient to hold the entire matrix (whose size is $O(n^2)$, assuming m/n bounded), otherwise the algorithm would be unrealistic.

In order to reduce the number of steps 3 and 4, we are currently studying a data structure where each line and each column are implemented by means of sorted linked lists of non-empty nodes. Scanning non empty nodes is then faster, however insertion of a new node in each list is slower.

III.4 Transitive closure :

After algorithm A has terminated :

i) for all $i \in N$, for all $j \in N$ such that $M(i,j) \neq \emptyset$ there is a path from i to j. The proof is given in appendix A.

ii) The complete paths from i to $j \in \{j \in N | \; M(i,j) \neq \emptyset\}$ are obtained by traversing the subgraph represented by column j : $\{M(i, j) | \; i \in N\}$, starting from node i (see Algorithm B in appendix).

IV CYCLES REDUCTION

Algorithms A and B ensure that all simple paths from i to j (paths without cycle) are obtained by traversing the subgraph represented by column j in matrix M. However when traversing this graph, we may encounter cyclic paths as illustrated by the following example :

$$\begin{tikzcd}
5 \arrow[rr] & & 1 \arrow[rr] \arrow[dr] & & 4 \\
& 3 \arrow[ur] & & 2 \arrow[ll] &
\end{tikzcd}$$

Figure 3 Example of graph

$$\begin{array}{c|c|c|c|c|c|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 2 & 2 & 2 & {}^{4}{}_{2} & \\
\hline
2 & 3 & 3 & 3 & 3 & \\
\hline
3 & 1 & 1 & 1 & 1 & \\
\hline
4 & & & & & \\
\hline
5 & 1 & 1 & 1 & 1 & \\
\hline
\end{array}$$

Figure 4 Matrix M with cycle

Let us solve the following query : "Give all the paths between 5 and 4"
$M(5,4) \neq \emptyset$. Then there exists at least a path.

Algorithm given B performs a depth-first search on the graph of column 4 :

1) Initially, the Departure Node (DN) is 5, the first Visited Node (VN) is 5.

2) Access M (5, 4) : node 1 is chosen.

DN = 1 VN = {5, 1}

3) 4 ∈ M (1, 4) ; one path has been found 5, 1, 4:

4) The next choice is node 2, since node 2 ∈ M (1, 4) :

DN = 2 VN = {5, 1, 2}

5) M (2, 4) = {3} :

DN = 3 VN = {5, 1, 2, 3}

6) M (3, 4) = {1}. 1 has been marked as already visited, so a cyclic path starting with node 1 is detected and node 3 is rejected. Backtracking will reject also nodes 2, 1 and 5.

In conclusion, a cyclic path is generated by Algorithm A and stored in M (i, j) which cycle is detected when traversing the graph.

Now, suppose we change the node labelling function so as to obtain the following permutation π_5 of 5 nodes :

$\pi_5(1) = 5$ $\pi_5(2) = 3$ $\pi_5(3) = 4$ $\pi_5(4) = 1$ $\pi_5(5) = 2$

$$\begin{tikzcd} 2 \arrow[rr] & & 5 \arrow[rr] \arrow[dr] & & 1 \\ & 4 \arrow[ur] & & 3 \arrow[ll] & \end{tikzcd}$$

Figure 5 Graph after permutation

	1	2	3	4	5
1					
2	5		5	5	5
3	4		4	4	4
4	5		5	5	5
5	1		3	3	3

Figure 6 Matrix M of the new graph

Clearly the cycle 5, 3, 4, 5, 1 has not been generated : M (5, 1) = {1} while in figure 4, M(1,4) = {4,2}.

Therefore a change in the node labelling function allowed to get rid of the cycle. We shall see below that in the general case we cannot find a node labelling function which allows to get rid of all cycles.

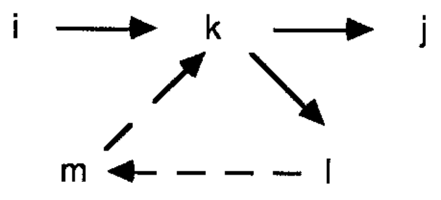

Figure 7a Elementary cycle

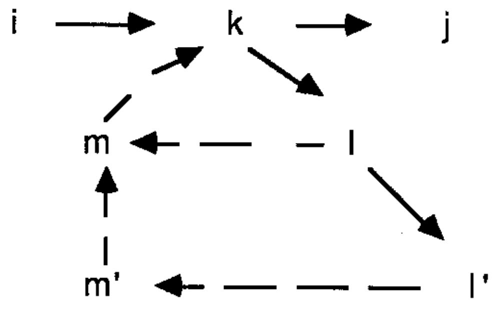

Figure 7b all cycles satisfying condition I

Let **c** = (k,l,...,m,...,k) be an elementary cycle in graph G (Fig 7a) satisfying the following conditions :

1) There exists a path from k to j, j ∉ **c**

2) for all m ∈ **c**, all paths from m to j go through k

Let **C** = ∪c be the union of such cycles, i.e. the set of nodes n which are on the same cycle as k (see example of Fig 7b) and such that all paths from m to j go through k

Lemma

If the permutation **π** of N is such that :

k = MAX (m | m ∈ C), then l ∉ M (k, j) after Algorithm A has terminated, otherwise l ∈ M (k, j), for all l, immediate successors of k on **c** ∈ **C**

Proof : see appendix C

Corollary

Given a path p : i...k...j, if k is an integer larger than the label of nodes

1) on the same cycle as k

2) and not connected to j, then we are ensured that Algorithm A will not produce the paths with a cycle starting with k.

Clearly, all cycles produced by Algorithm A are not eliminated, as illustrated by the example on Fig 8

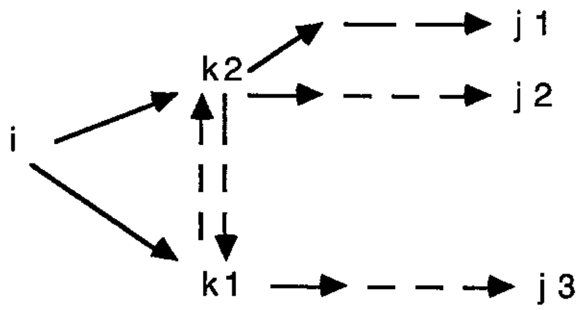

Figure 8 All cycles are not eliminated

if $k_1 > k_2$, the following paths are eliminated :

$i\ k_1 \ldots k_2 \ldots k_1 \ldots j_1$

$i\ k_1 \ldots k_2 \ldots k_1 \ldots j_2$

but the cyclic path : $i\ k_2 \ldots k_1 \ldots k_2\ j_3$ remains.

The above lemma does not suggest a simple way of minimizing the number of cycles produced by algorithm A, although the following node labelling function might in most cases reduce the number of cycles.

Let O(i) be the outdegree of node $i \in N$.
Let Π be a permutation of N such that nodes are sorted by increasing outdegree :

Π : $i < j$ iff $O(i) < O(j)$

V. STRATEGIES FOR IMPLEMENTING THE GENERALIZED TRANSITIVE CLOSURE

We consider the extension of relational systems by including an operator for computing the transitive closure of :

$g(R) = \Pi_P\ (\sigma_F (R \times R))$

We also assume that queries may involve some aggregate calculus as well as some memorization about the path followed when computing the transitive closure. As examples of aggregate calculus, on the FLIGHTS relation, one may be interested in selecting the flights from L.A. to Roma, the cheapest or with a minimum number of connections, ...

We look at strategies based on the implementation of the transitive closure by means of two algorithms :

a) Algorithm A of section III
b) A regular graph traversal algorithm [Puc87, Agr87]

Both algorithms assume the existence of data structures representing the graph associated to the transitive relation R. Such data structures allow fast transitive closures but imply more complicated relation updates.
In the second algorithm, all we need is the graph transition matrix N (i, j), while in the first case the basic data structure is the matrix M (i, j) produced by algorithm A (see section III).

V.1 Queries

In the general case a selection σ_F on R^2 implies that paths will be selected on G(R,A,B) according, not only to the edges label value, but also according to the nodes label value. In other words, paths in the graph can be eliminated because a condition is not satisfied for

1) some edge on the path,

2) some node on the path

3) the aggregate function (e.g. +,*,...) on the values of the edges of the path.

In general, the verification of condition 3) requires scanning all solutions to the query, i.e. traversing all paths associated to the transitive closure.

As for as selection conditions 1) and 2) are concerned, whether the selection and the closure operators are transposed or not will give various strategies.

Before reviewing theses strategies, let us specify the selection operator σ_F :

Assume F may be written as

$F = F_{A1}$ <u>and</u> F_{B2} <u>and</u> F' <u>and</u> $(A2 = B1)$

where :

1) To each value of A or B corresponds a node in G (R, A, B).

2) In the schema of the product R X R, A1, B1 (A2, B2) correspond to the first (second) instance of attribute names A, B.

3) F_{A1} (F_{B2}) contains all relational predicates involving A1 (B2) and none of the others (condition on the node)

4) F' contains all relational predicates involving other attributes (condition on the edges).

For the sake of simplicity, we assume there is no projection condition : all attributes are projected.

V.2 Strategies based on Algorithm A

V.2.1 First strategy :

A natural and efficient strategy consists in selecting the edges to be eliminated (because of the selection condition $\sigma_{F'}$), then two cases may happen in the resulting graph :

a) no node is eliminated : this might occur in some applications for which the number of tuples associated to an edge is large. Then

1) either access directly M(i,j) if only the transitive closure is required,

2) or traverse the graph (Algorithm B)

b) some nodes must be eliminated. Then mark them and apply Algorithm B in all cases.

This strategy is presented in the following algorithm :

<u>Algorithm C</u>

```
begin
        $R1 \leftarrow \pi_{A,B}(\sigma_{F'}(R))$
        $R2 \leftarrow R - R1$
        if $R2 = \emptyset$
        then
                $K \leftarrow \emptyset$
        else
                $R3 \leftarrow \pi_A(R2) \cup \pi_B(R2)$
                $K \leftarrow \varphi_n(R3)$
        end if

        */ $K \in \{1,\ldots,n\}$ is the subset of nodes to be bypassed when traversing the graph /*

        $I \leftarrow \varphi_N(\sigma_{FA1}(R))$
        $O \leftarrow \varphi_N(\sigma_{FB2}(R))$

        */ I and $O \subset \{1,\ldots,n\}$ are the subsets of entry (output) nodes in the graph /*

        LOOK–PATH (I, O, K)
end
```

LOOK–PATH (Algorithm B, see Appendix B) traverses the graph and looks for all paths from $i \in I$ to $o \in O$, such that $k \in K$ is forbidden.

Notes

1) We have assumed that in all cases we are interested in the whole path. If just the transitive closure is desired (and $K = \emptyset$) then it is given by $M(i,o)$, $i \in I$, $o \in O$.

2) For the sake of simplicity there is no aggregate function in the computation of PATH.

3) The calculation of R2 (one selection, one projection and one difference) can be reduced to a single selection/projection if R is in N1NF (see for example [JV87]).

V.2.2. Second strategy :

It consists in traversing the graph and checking at each edge traversal the selection condition $\sigma_{F'}$: if the tuple does not satisfy the condition, the path is abandonned. (Algorithm B).

V.2.3. Third strategy :

Finally, the third strategy applies the selection first : $R'' \leftarrow \sigma_{F'}(R)$, recomputes matrix M(i,j) for R' (Algorithm A), then accesses M(i,j) or traverses the graph (Algorithm B).

In practice, the two last strategies are too expensive in time, and one should prefer the first strategy even though the selection condition $\sigma_{F'}$ is such that the graph G(R,A,B) is much altered. However in the case where G(R', A, B), (where R' = $\sigma_{F'}$(R)) is expected to have a small number of nodes, one might apply the second strategy (recompute M(i,j)).

V.3 Strategies based on a regular graph traversal algorithm

If G(R,A,B) is represented by the nxn matrix N(i,j), then there are three strategies analoguous to that of section V.2 :

1) select and mark nodes to be eliminated, then traverse the graph. This is exactly Algorithm C, except that LOOK-PATH is implemented by means of Algorithm D (see Appendix D).

2) when traversing each edge of the graph (Algorithm D), condition $\sigma_{F'}$ is checked.

3) one applies first selection $\sigma_{F'}$, then N(i,j) is recomputed, then the graph is traversed.

It is worth noting that in all cases :

1) the whole graph is traversed: in the previous section one traverses only the graph corresponding to one column (one end point) which is eventually reached,

2) backtracking in this graph is done not only because of cycles, but especially because we do not come up to a desired end point,

3) graph traversal is required even if the query is a simple transitive closure and does not imply memorizing information about the path followed.

Of course, the more the query is selective, the better the third strategy is.

Algorithms for these three strategies are almost identical to that of section V.2. The only difference lies in the use of a different graph traversal algorithm, (Algorithm D, see Appendix D) used also for simple transitive closure, and taking as an entry the whole graph represented by the transition matrix N(i,j).

VI CONCLUSION

We have introduced in this paper a new algorithm for implementing a generalized transitive closure operator as defined in [SS88]. This algorithm is a mixture of a Warshall like algorithm [War62] and graph traversal.

We reviewed then a few strategies for implementing the transitive closure. Those strategies use as a primitive the algorithm introduced above or any regular graph traversal algorithm [Puc87, Agr87]. A performance comparison of these strategies with both algorithms is under study.

Whether the new algorithm performs better than a regular graph algorithm will depend on the application.

The new algorithm presents the following advantages :

1) a fast transitive closure, while with a graph traversal algorithm, the transitive relation is represented by its associated graph and computing the transitive closure requires traversing the graph which may be time consuming

2) as a regular graph traversal algorithm, but at a lower cost :

   1) the choice of paths on the basis of aggregate functions calculus

   2) to memorize information on the path followed.

With this new algorithm, for some queries, graph traversal is even the olny possible way of computing the transitive closure.
Futhermore computing the data structure required by the new algorithm has been predicted to be in the worst case time consuming unless the relation is of limited size. This comes from the fact that all path associated to the transitive relation are partially precomputed in order to render any query fast at run-time.
We are currently studying another implementation of this algorithm in order to reduce the time complexity of the data structure construction and undertaking a measurement experiment for typical applications in order to get an evaluation of the average case time complexity and infirm or confirm the worst case prediction. We believe that in some situations even with large relations, the data structure construction should be fast enough. In particular, we are studying the case of relations whose associated graph is a set of clusters sparely connected. An example of such a situation would be an intercity communication network whose intracity communications would correspond to clusters.

REFERENCES

[Agr87] R. Agrawal, "Alpha : An extension of relational algebra to express a class of recursive queries", Proceedings of the 3rd Data Engineering Conference, Los Angeles, CA, February 1987.

[AJ87] R. Agrawal, H.V. Jagadish, "Direct Algorithms for computing the Transitive Closure of Database Relations", Proceedings of the 13th International VLDB conference, Brighton, England, September 1987, pp 255-266.

[BR86] F. Bancilhon, R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies", Proceedings of the ACM SIGMOD 86 conference

[GS87] G. Gardarin, E. Simon, "Les systèmes de gestion de bases de données déductives", T.S.I., Vol 6, N° 5, 1987, pp. 347-382.

[JV87] Jules VERSO, "VERSO : A database machine based on N1NF relations", INRIA report n° 523, May 1986.

[Lu87] H. Lu, "New strategies for computing the Transitive Closure of database relations", Proceedings of the 13th VLDB Conference, Brighton, 1987.

[LMR87] H. Lu, K. Mikkilineni, J.P. Richardson, "Design and Evaluation of Algorithms to compute the Transitive Closure of a Database Relation", Proceedings of the 3rd International Data Engineering Conference, Los Angeles,CA, February 1987.

[Nau87] J.F. Naughton, "One-sided Recursion", Proceedings of the 6 th ACM, PODS Conference, 1987.

[Puc87] P. Pucheral, "Les structures d'adjacence : Un support adapté au traitement des règles récursives linéaires", 3ème Journées Nationales des Bases de Données, Port-Camargue, 1987.

[RHD86] A. Rosenthal, S. Heiler, U. Dayal, F. Manola, "Traversal recursion : A practical approach to supporting recursive applications", Proceedings of the ACM SIGMOD Conference, 1986.

[SS88] S. Sippu, E.S. Soininen, "A generalized transitive closure for relational queries", Proceedings of the 7th ACM PODS Conference, Austin, Texas, March 1988.

[Sch83] L.Schmitz, "An Improved Transitive Closure Algorithm", Computing, Vol 30, 1983, pp 359-371.

[War62] S. Warshall, "A theorem on boolean matrices", J.A.C.M. 9,1, 1962.

APPENDIX A : Proof ot the validity of algorithm A :

It is obvious that as in the case of Warshall's algorithm, Algorithm A adds only arcs that should be added, i.e. if it sets :

$$M(i,k) \leftarrow M(i,k) \cup M(i,j)$$

then it adds one (or several) path(s) from i to k through j.

The only thing deserving proof is that algorithm A adds <u>all</u> the paths (i, j) existing in the graph.

This can be proved by induction on the length of any path from a to b. We just give a sketch of the proof :

If the path length is one, then the edge (a, b) is already present : $b \in M(a, b)$.

Assume that for all paths of length $l > 1$, Algorithm A adds the path (a, b) and we have :

$$s(a) \in M(a,b)$$

where s(a) is the immediate successor of a on the path (a, b).
Now take a path of length $l+1$ and let m be the column that algorithm A processes last : m is the highest numbered node on the path :

$$a \ldots m \ldots b$$

Assume $m \neq a, b$ (The cases $m = a$ or $m = b$ can be treated similarily). We are now leaved with two shorter paths (a, m) and (m, b). By the inductive hypothesis, they have already been added to the graph by algorithm A, when it is processing column m, since all other columns corresponding to nodes of the path have been already scanned.

More specifically, let n(p) be the column that algorithm A processes last when adding the path (a,m) (the path (m, b)) :

$$a, s(a), \ldots, n, \ldots, m, \ldots, p, \ldots, b$$

When processing column n, we insert s(a) in M(a,m) :

$$M(a,m) \leftarrow M(a,m) \cup M(a,n) \quad \text{iff } M(a,n) \neq \emptyset \text{ and } M(n,m) \neq \emptyset.$$

Indeed, $M(a, n) \neq \emptyset$ since there is a path from a to n, and
$M(n, m) \neq \emptyset$ since there is a path from n to m, therefore :
$M(a, m) \neq \emptyset$ (1)

Similarily, when processing column p, we have :

$$M(m,b) \leftarrow M(m,b) \cup M(m,p)$$

then, $M(m, b) \neq \emptyset$ (2)

Finally, when processing column m and line a we insert s(a) in M(a,b). Indeed we have :

$$M(a,b) \leftarrow M(a,b) \cup M(a,m)$$

since by equations (1) and (2), M(a, m) and M(m, b) are not empty. Then we have indeed added the path (a, b) of length n+1.

APPENDIX B

In this appendix we give a trivial algorithm for graph depth-first traversal, when the graph is represented by matrix M (i,j).

Let A be a n-vector whose value is initially :

A (i) = True, if i is the destination node of an edge which does not satisfy the selection condition $\sigma_{F'}$: i $\in \psi_N$ (R3), (see Algorithm C, section V.2.1)

A (i) = False, otherwise

During the graph traversal, each time a node, say i, is visited, A(i) is set to TRUE to avoid cycles.

Procedure LOOK-PATH (I, O, A)

```
        begin
            for all i ∈ I, j∈ O
                if M (i, j) ≠ ∅
                then
                    if i ≠ j
                    then
                        A (i) = True
                    endif
                    PATH = < i >
                    FIND—PATH (i, j, A, PATH)
                    A(i) = False
                endif
            endfor
        end
```

PATH is a (set of) finite sequence(s) of nodes starting with i and finishing with j.

I is the set of departure nodes, O is the set of arrival nodes.

Procedure FIND-PATH (i, j, A, PATH)

```
begin
        for all k ∈ M (i, j)
         do
            if k = j
            then
                output (PATH + < j >)
            else
                if A(k) = False
                then
                    A (k) = True
                    FIND—PATH (k, j, A, PATH+ < i >)
                    A (k) = False
                endif
            endif
        endfor
end
```

APPENDIX C

Let us denote by π a permutation of $N = \{1,2,\dots,n\}$ and by $M(G)$ the matrix associated to G. $G\pi$ denotes the graph obtained from G by appling to its n nodes the permutation π.

Lemma :

If the permutation π of N is such that :

$k = \text{Max}\,(m \mid m \notin C)$, then $1 \notin M(k,j)$ after Algorithm A has terminated, otherwise $1 \in M(k,j)$, for all 1 immediate successors of k on cycles $c \in C$.

Proof :

Let $p = \text{MAX}\,(m \mid m \in C)$. There are two cases :

1) $p = k$:

Then column k is processed after all other columns $m \in C$.

a) For all $m \in C$, $m \neq k$, $M(m,j) = \emptyset$, before Algorithm A processed column k.

Indeed before the latter step, the path m...k...j has not been detected yet and by assumption, there is no other path from m to j. $M(m,j)$ is modified when processing column k.

b) Before Algorithm A starts, $M(k,j) = \{j\}$ and of course $1 \notin M(k,j)$.

Then, the only way of modifying $M(k,j)$ is, when processing column m and line k through the step :

$$M(k,j) \leftarrow M(k,j) \cup M(k,m), \quad \text{if } M(m,j) \neq \emptyset$$

Observe first that $m \neq k$: when processing column k, line k is not modified.
Then there are two cases :

i) $m \in C$, then $M(m,j) = \emptyset$, by argument a) above; therefore $M(k,j)$ is not modified.

ii) $m \in C$, then assume $M(m,j) \neq \emptyset$: there is a path between m and j.

Then,
-either this path goes through k, which is in contradiction with the assumption that $m \in C$,

-or, if $1 \in M(k,m)$, then there is a path from 1 to m and therefore there exists a path from 1 to j, which does not go through k, which is in contradiction with the assumption.

2) $p \neq k$:

We shall show in three steps that $1 \in M(k,j)$ after column p has been processed :

a) We first show that $1 \in M(k,p)$:

Before processing column p, all paths on C from k to p have been detected. For all m on these paths, we have :

$m < p$
$1 \in M(k, m)$
$M(m, p) \neq \emptyset$.

Therefore there exists such an m column which, when processed, provides the following update of $M(k, p)$:

$$M(k, p) \leftarrow M(k, p) \cup M(k, m)$$

then $1 \in M(k,p)$.

b) We then show that when starting to process column p, $M(p, j) \neq \emptyset$:

Using the same argument as in a) a path has been detected between p and k on C just before processing p. Since there is an edge from k to j, there is a path from p to j and before starting processing p, $M(p, j) \neq \emptyset$.

c) Finally, when processing column p, on line k, we have :

$$M(k,j) \leftarrow M(k,j) \cup M(k,p), \text{ if } M(p, j) \neq \emptyset$$

then, from a) and b) :

$$1 \in M(k, j).$$

APPENDIX D

Algorithm D

The following graph traversal algorithm computes a path in a graph represented by its adjacency matrix N (i, j). It is very similar to Algorithm B.

LOOK–PATH (I,O,A)

```
begin
        if i ≠ j
        then
                A(i) = True
        endif

        PATH = < i >

        FIND–PATH (i, j, A, PATH)

        A(i) = False

end
```

FIND–PATH (i, j, A, PATH)

```
begin

        for all k ∈ {1, n}
         do
                if N (i, k) = 1
                then
                        if k = j
                        then
                                output (PATH + < k >)
                        else
                                if A(k) = False
                                then
                                        A(k) = True
                                        FIND–PATH (k, j, A, PATH + < i >)
                                        A(k) = False
                                endif
                        endif
                endif
        endfor
end
```